Porting the CORTEX RTOS to the MSC8101

Application Note

by
Mihai Fecioru,
Cristian Zamfirescu
and
Emilian Medve

AN2191/D
Rev. 0, 10/2001

STARCORE

MOTOROLA

CORTEX is a trademark of Australian Real Time Embedded Systems (ARTESYS)

Star*Core is a trademark of Motorola, Inc.

Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters which may be provided in Motorola data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals” must be validated for each customer application by customer’s technical experts. Motorola does not convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support life, or for any other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part.
Motorola and Ⓜ are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.
All other tradenames, trademarks, and registered trademarks are the property of their respective owners.

How to reach us:

USA/EUROPE/Locations Not Listed: Motorola Literature Distribution; P.O. Box 5405, Denver, Colorado, 80217.
1–303–675–2140 or 1–800–441–2447

JAPAN: Motorola Japan Ltd.; SPS, Technical Information Center, 3–20–1, Minami–Azabu, Minato–ku,
Tokyo 106–8573 Japan. 81–3–3440–3569

ASIA/PACIFIC: Motorola Semiconductors H.K. Ltd., Silicon Harbour Centre, 2 Dai King Street,
Tai Po Industrial Estate, Tai Po, N.T., Hong Kong. 852–26668334

Technical Information Center: 1–800–521–6274

HOME PAGE: http://www.motorola.com/semiconductors/

© Copyright Motorola, Inc., 2001

Abstract and Contents

Application software targeted for today's digital signal processors is becoming more complex. DSPs are now incorporated with numerically intensive algorithms and must perform complex system control and communication protocols previously relegated to general-purpose microprocessors. When a complicated control is mixed with DSP software, a real-time kernel is required. In order to achieve a short time to market, porting an existing RTOS may prove to be a good choice. This document describes the porting of the CORTEX RTOS to the Motorola MSC8101 DSP based on the StarCore SC140 core.

This application note is addressed to StarCore SC140 RTOS developers and system programmers.

1 Introduction 1

1.1 RTOS Requirements for DSPs 1
1.2 Development Tools 1

2 MSC8101 Hardware Platform 2

2.1 General Description 2
2.2 Features Used by the RTOS 2
2.2.1 StarCore SC140 Overview 2
2.2.2 Interrupt Programming 4
2.2.3 Periodic Interrupt Timer 5

3 CORTEX Overview 5

3.1 Platform-Independent Layer 6
3.1.1 Interrupt Manager 6
3.1.2 Task Manager 7
3.1.3 Memory Manager 8
3.1.4 System Clock 9
3.2 HAL—Platform-Dependent Layer 9

4 Porting Details 13

4.1 Platform Initialization 13
4.1.1 Boot Code 13
4.1.2 Linker Command File 13
4.2 Interrupt Management 13
4.2.1 The Interrupt Table 13
4.2.2 Default LISR Dispatcher 14
4.2.3 The LISR and HISR Stacks 16
4.2.4 Activating/Deactivating Interrupts 17
4.2.5 Atomic Operations 17

4.3 Task Management 18
4.3.1 Task's Stack 19
4.3.2 Stack Frame Tracing 20
4.3.3 The Idle Task 20
4.4 System Clock 20
4.4.1 Clock Initialization 20
4.4.2 Clock LISR 21
4.5 Memory Management 21
4.6 Building the System 21
4.6.1 Makefiles 21
4.6.2 CodeWarrior Project 23

5 Testing 24

6 Results 24

7 Conclusions 25

8 References 26

1 Introduction

1.1 RTOS Requirements for DSPs

The design of a StarCore SC140 real time operating system (RTOS) will support features needed for real time Digital Signal Processing applications. Required features include the following:

- Multitasking with preemptive task scheduling.
- Scheduling:
  - — Preemptive.
  - — Round Robin (within a priority).
  - — Time-Sliced scheduling within the same priority level.
  - — Fixed or dynamically changeable task priorities.
- Support for statically created tasks, dynamically created tasks and dynamic downloading of tasks.
- Intertask communication via semaphores, messages and queues.
- Semaphores for synchronizing tasks with internal or external events.
- Support for SC140 timer management for timer based operations.
- Deterministic memory management support.
- Support for timeout features in the API.
- Support for Debugger Awareness
- Device resource management.
- Fast and efficient context switching.
- Small RAM/ROM requirements.
- Support for Interrupt Service Routines (ISR).
- Optional calculation of runtime processor loading. This information must be available to either an internal or external application (such as a performance monitor).

1.2 Development Tools

The development tools used include the Metrowerks StarCore Enterprise C compiler version 1.1 along with the Metrowerks CodeWarrior version 4.1 for StarCore R1.0.

Changes to the platform-independent layer of CORTEX were tested using GCC compiler version 2.95.3 on a Sun SPARCstation 20 running Solaris 7.

The project was developed on a Microsoft Windows 2000 Professional host. The hardware tests were performed on an MSC8101 Application Development System (ADS).

2 MSC8101 Hardware Platform

2.1 General Description

The Motorola MSC8101 is a versatile, one-chip integration of a high-performance SC140 core, large on-chip memory (0.5 MB), a communications processor module (CPM), a very flexible system interface unit (SIU), and a 16-channel DMA controller.

The main modules of the MSC8101 are:

- SC140 core: a powerful, latest-generation DSP core capable of up to 1200 true DSP MIPS or 3000 RISC MIPS with a 300 MHz clock.
- System Interface Unit (SIU): contains a clock synthesizer, reset controller, two interrupt controllers, real-time clock register, periodic interrupt timer, hardware bus monitor and software watchdog timer.
- Communications Processor Module (CPM): contains a set of communication controllers supporting various protocols and interfaces.

2.2 Features Used by the RTOS

2.2.1 StarCore SC140 Overview

The SC140 digital signal processing (DSP) core is the first implementation of an innovative architecture that addresses the key market needs of next-generation DSP applications. This flexible, programmable DSP core facilitates the development of computation-intensive communication applications by providing high performance, low power, efficient compilation, and high code density. The SC140 core efficiently deploys a novel variable-length execution set (VLES) execution model, maximizing parallelism by allowing multiple address generation and data arithmetic logic units to execute multiple operations in a single clock cycle.

2.2.1.1 Data Arithmetic Logic Unit

The Data ALU (DALU) performs arithmetic and logical operations on data operands in the SC140 core. The data registers can be read or written to memory as 8-bit, 16-bit, or 32-bit operands. Two 64-bit wide data buses (XDBA and XDBB) support the transfer of several operands on a single access. The source operands for the Data ALU, which can be 16, 32, or 40 bits, can be either from data registers or immediate data. The results of all DALU operations are stored in the data registers. All DALU operations are performed in one clock cycle. Up to four parallel arithmetic operations can be performed in each cycle. The destination of every arithmetic operation can be used as a source operand for the operation immediately following, without any time penalty.

The components of the DALU are:

- A bank of sixteen 40-bit data registers
- Four parallel ALUs, each ALU containing a Multiply And Accumulate (MAC) unit and a Bit-Field Unit (BFU) with a 40-bit barrel shifter
- Eight data bus shifter/limiter circuits to allow saturation of a four-word transfer over each of the XDBA and XDBB buses in a single cycle

2.2.1.2 Address Generation Unit

The AGU is one of the execution units in the StarCore SC140 core. The AGU performs effective address calculations using the integer arithmetic necessary to address data operands in memory, and also contains the registers to generate the addresses. The AGU operates in parallel with other chip resources to minimize address generation overhead. The AGU also generates change-of-flow program addresses and updates the stack pointer (SP) whenever needed.

The major components of the AGU are:

- Eight address registers (R0–R7])
- Eight alternative address registers (R8–R15]) or eight base address registers (B0–B7])
- Two stack pointers (NSP, ESP), only one of which is active at a time (SP)
- Four offset registers ([N0–N3])
- Four modifier registers ([M0–M3])
- A Modifier Control Register (MCTL)
- Two address arithmetic units (AAU)
- One-bit mask unit (BMU)

An important StarCore facility is the existence of two different stack pointers: NSP (Normal Stack Pointer) and ESP (Exception Stack Pointer). The NSP is used when the core is in the normal processing mode. The ESP is used in the exception mode by the OS and interrupts. When an exception occurs, the core switches to the ESP and uses the exception stack for saving registers and allocating local variables and subroutine calls.

The SP register points to the next unoccupied cell. The SC140 stack has eight byte-wide cells which allow it to implement up to two PUSH/POP instructions in a single execution set. If two PUSH instructions are included in a single execution set, the first PUSH must use an even register operand, and the second one must use an odd register operand. A PUSH instruction always pushes one 32-bit register into the stack. Any execution set that includes one or two push instructions increments the stack pointer by eight. In the case of a single PUSH, a single operand is written to memory while the adjacent memory location remains unchanged.

2.2.1.3 Status Register (SR)

The status register is one of the StarCore control registers. Its bits reflect and control the status of different parts of the core. The most important bits for this project are:

- SR[21:23]—the Interrupt Priority Level (IPL) bits reflect the current interrupt priority level of the core. Only non-maskable interrupts or interrupts with an IPL higher than the current interrupt priority level can interrupt the core.
- SR[19]—the DI (Disable Interrupts) bit is set when non-maskable interrupts are serviced, regardless of the IPL value, which remains unchanged.

2.2.1.4 Exception Processing

When an interrupt occurs, the current task is stopped, PC and SR are pushed onto the stack and program execution continues from the exception vector address associated with the selected exception, which is calculated as follows:

- Bits[31:12]—the VBA register which points to the interrupt vector base
- Bits[11:6]—the interrupt offset in the interrupt table provided by the Program Sequencer (PSEQ) or by the Programmable Interrupt Controller (PIC)

- Bits[5:0]—always zero, allowing 64 bytes for each entry in the interrupt vector

The PIC is illustrated in Figure 1.

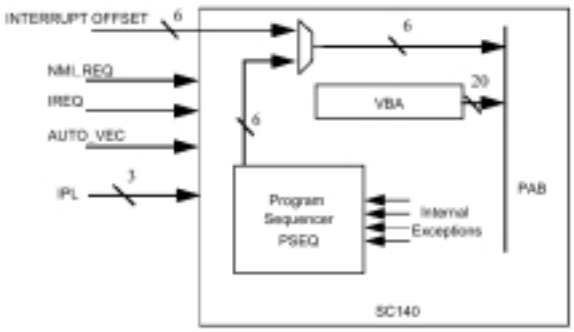

Figure 1. StarCore PIC Interface

When an interrupt is serviced, the SR register is saved and the IPL bits are set to the interrupt priority value. This automatically disables all the maskable interrupts with priorities lower or equal to the IPL value. All other maskable interrupts and the non-maskable interrupts will be serviced.

2.2.2 Interrupt Programming

The MSC8101 interrupt scheme includes the following interrupt controllers:

- Programmable Interrupt Controller (PIC), which operates in the SC140 core.
- SIU-CPM interrupt controller (SIC), which generates interrupt requests to the PIC.
- External SIU-CPM interrupt controller (SIC_EXT), which generates interrupt requests to an external host CPU.

Only the PIC and the SIC are used by the CORTEX RTOS.

2.2.2.1 Programmable Interrupt Controller (PIC)

The MSC8101 PIC is a peripheral module that services interrupt requests (IRQs) and non-maskable interrupts (NMIs) received from MSC8101 peripherals and I/O pins. It has 8 asynchronous edge-triggered NMI inputs and 24 asynchronous edge-triggered/level-triggered IRQ inputs. Each IRQ can be programmed to a priority level between 0 and 7, where 0 disables the interrupt.

When the PIC detects an IRQ on one or more of its inputs, it arbitrates each IRQ according to its priority level and location and generates the following:

- An IRQ signal to the SC140 core, indicating that an IRQ input has requested interrupt service by the SC140 core.
- A three-bit IPL signal indicating the priority of the IRQ.
- A six-bit offset in the predefined VAB, determined by the location of the IRQ.

StarCore's AUTO_VEC signal is not used in the MSC8101.

PIC programming is done using the following state and control registers:

- Edge/Level-Triggered Interrupt Priority Registers (ELIRA-ELIRF): six 16-bit read/write control registers used for controlling interrupt priority level and trigger mode for each interrupt. Each of the 24 IRQs has four bits associated in one of these six registers, three for the priority level and one for the trigger mode.

- Interrupt Pending Registers (IPRA, IPRB): two 16-bit read/write registers used for monitoring pending interrupts and resetting edge-triggered interrupts. There is one bit for each interrupt source (IRQ or NMI) indicating its current state (pending or not pending).

2.2.2.2 SIU – CPM Interrupt Controller (SIC)

This controller receives various interrupt requests (from SIU and CPM sources or DMA and external sources) and routes them to the core via the PIC. Entry 48 in the PIC interrupt table (IRQ16) is the SIC interrupt request. The SIC interrupt vector number can then be read from the SIVEC register.

The SIC has some flexibility in assigning relative priorities for interrupt requests, but not as much as the PIC. Relative priorities can only be changed within groups of interrupt requests. In addition, a highest priority interrupt can be defined.

SIU registers used in this project include:

- SIU High Interrupt Pending Register (SIPNR_H), SIU Low Interrupt Pending Register (SIPNR_L): these 32-bit registers contain one bit for each interrupt source to indicate if an interrupt request is pending.
- SIU High Interrupt Mask Register (SIMR_H), SIU Low Interrupt Mask Register (SIMR_L): these 32-bit registers contain one bit for each interrupt source which can be used to mask the interrupt requests.
- SIU Interrupt Vector Register (SIVEC): contains the interrupt vector of the interrupt request with the highest priority that is pending and unmasked.

2.2.3 Periodic Interrupt Timer

The SIU module contains two timers, the Periodic Interrupt Timer (PIT) and the time counter.

The PIT is used in CORTEX as a system clock. It consists of a 16-bit counter (PITR) that decrements to zero when loaded with a value from the Periodic Interrupt Timer Count Register (PITC). When the counter reaches 0, an interrupt request is generated, the value of the PITC is loaded in the counter and the process repeats. The interrupt request is reported via the SIC (vector 17 in the SIC interrupt table).

The timeout period of the periodic interrupt timer ranges from 122 µs to 8 seconds. The PIT can be controlled using the Periodic Interrupt Status and Control Register (PISCR).

The input clock for the PIT can be derived either from Baud Rate Generator 1 (BRG 1) or an external source. The frequency of the input clock must be 8192 hertz, because the time counter, which uses the same clock source, requires this frequency for proper operation.

3 CORTEX Overview

The starting point for this project was version 1.01.00 of the CORTEX RTOS. CORTEX is an abbreviation for COncurrent Real-Time EXecutive.

- CONCURRENT—the system is multi-tasking. Task scheduling is preemptive, event-driven and priority-based. A comprehensive set of inter-task synchronization and communication primitives is supported.
- REAL-TIME—the kernel is designed to be used for embedded applications; it responds immediately to internal and external events.

- EXECUTIVE—CORTEX is implemented as a runtime library to be linked with an application and downloaded to a hardware platform as a single entity.

CORTEX is presented as a collection of libraries (the OS), and any application that is run on it is an object file. At link time, only those references required by the application object file are provided from the libraries. For example, if an application does not use semaphores, the linker does not include those files from the libraries. To ensure that the linker can include and exclude the appropriate system features, each feature in the C source code must be written in a separate file.

3.1 Platform-Independent Layer

3.1.1 Interrupt Manager

The interrupt management subsystem is the foundation on which the CORTEX kernel is based. It contains two major components—a hardware interrupt manager and a software interrupt manager.

3.1.1.1 Hardware Interrupt Manager

The hardware interrupt manager recognizes two interrupt control models:

- Interrupt mask-based model—interrupt sources are controlled by an interrupt mask. In this model, the CPU provides a special register in which every bit corresponds to a particular interrupt source. Setting and clearing an individual bit enables or disables the corresponding interrupt source.
- Interrupt priority-based model—each interrupt source is assigned an interrupt priority level. The CPU can be switched into different interrupt levels. Increasing the CPU's interrupt level disables all interrupt sources of the same or lower priority.

CORTEX provides two types of hardware interrupt handlers:

- Low-level interrupt service routine (LISR). By definition, the LISR is a hardware interrupt service routine that is activated by an interrupt dispatcher. An ordinary C function can be registered as a LISR, and no assembler programming is required. Each function receives two arguments—an interrupt vector number and a context. The LISR is not allowed to use any kernel services except `sfti_Trigger()` to trigger software high-level interrupt service routine (HISR). (The primary reason for this restriction is that the kernel is not protected against hardware interrupts.) LISRs can be nested, but the same LISR cannot be interrupted by itself (i.e., it is not reentrant). Each LISR can be assigned its own stack.
- Direct interrupt service routine (DISR). DISRs are activated directly by the interrupt controller. No dispatching overhead is involved, but the DISR must preserve the interrupted context. (This can usually be done via assembler code.) In addition, DISRs are not allowed to call any kernel services. Calling `sfti_Trigger()` would not harm the kernel, but the HISR would not be activated after returning from the DISR. This is because an HISR can only be activated after a hardware interrupt has been serviced by a dispatcher or an application task has entered kernel protection mode.

The hardware interrupt manager supports both a primary and secondary interrupt vector table. The primary interrupt table is used by the CPU. Each entry in this table (except the reset vector) refers the CPU to the appropriate entry in the secondary interrupt table. The primary interrupt table is never modified, and is usually loaded in ROM. The entries in the secondary interrupt table are the vectors to the actual interrupt service routines and can be modified when a new ISR is registered.

3.1.1.2 Software Interrupt Manager

The software interrupt manager simulates an interrupt controller. The simulated interrupt controller consists of 32 interrupt sources controlled through an interrupt enable/disable mask and a global interrupt flag. The global interrupt flag is a counter that is incremented by one when interrupts are disabled and decremented when interrupts are enabled. Each bit in the interrupt enable/disable mask controls an individual interrupt source. For each bit that is set, the corresponding interrupt source is enabled.

Each active interrupt vector has an associated high-level interrupt service routine (HISR). A LISR triggers a HISR by calling the `sfti_Trigger()` routine. All triggered HISRs are activated by a hardware interrupt dispatcher after all pending LISRs have been processed. The pending HISRs are activated one by one with the highest priority HISR activated first. One HISR cannot be interrupted by another HISR, but it can be interrupted by a LISR or DISR.

3.1.2 Task Manager

The ability to control many independent tasks in a flexible and an efficient way is an important feature of the kernel. A task is an independent thread of execution with a set of individual resources assigned to it, which is constantly competing with other tasks for system resources.

CORTEX's API defines two separate sets of interfaces to manage tasks—a task manager and a thread manager. The task manager component is a restrictive wrapper around the thread manager. The task manager only utilizes a subset of the available services. Internally, the kernel uses the thread manager because it provides a more powerful and comprehensive set of services.

A task can exist in one of the following states:

- Running—the task is currently occupying the CPU
- Ready—the task can be scheduled to CPU
- Deferred—the task has been created, but has not yet been started
- Terminated—the task has been terminated, but still continues to occupy individual resources
- Blocked—the task is waiting for an event with no timeout interval provided
- Waiting—the task is waiting for an event and a timeout interval has been provided
- Sleep—the task cannot be scheduled to the CPU until a specified interval of time expires
- Suspended—the task was suspended by another task
- Free—the task's slot and control block are available for a new task

A CORTEX task owns a set of individual resources that are allocated when the task is created. These resources include:

- task slot—task slot number is an index within the task table
- control block—holds all of the specific information for a task
- stack space—used to allocate automatic variables as well as procedural and interrupt stack frames
- private memory segment—this memory segment is private to every task
- environment—a means of communication between the kernel and the task

Each task has an associated priority number within a range of 1 to 62. (The thread manager also supports priority values 0 and 63, but these are reserved for internal use) and a scheduling policy. CORTEX provides four different scheduling policies: time-sharing, round robin, round robin & time slicing, another policy.

Two tasks are always created on kernel startup—the Idle task (`thrd_IdleThread()`) and the Main task (`Cortex()`). The Idle task is the lowest priority task (level 0) and never blocks (i.e., is always in either the ready or running state). This guarantees that the task scheduler always has a task to run. In addition, as the lowest priority task, the Idle task only runs if all other tasks are blocked. The Idle task is created as a demon, meaning that it is automatically deleted when the last non-demon task terminates.

The Main task calls an application's main routine. It is created automatically and has the highest priority. The `crtx_Main()` procedure is called in the Main task context and must be the starting entry point of every CORTEX-based application.

The task switch is performed by a single function, `thrd_Scheduler()`. This function is called only after all pending hardware and software interrupts are serviced. The task scheduler can only be called by one complementary pair of functions, `sfti_GlobalForbid()` and `sfti_GlobalPermit()`.

The `sfti_GlobalForbid()` function increments a global variable called `sfti_Environ_g.Global` which is the global disable flag for software interrupts. HISRs are disabled when this variable has a value greater than 0. The `sfti_GlobalPermit()` function services pending software interrupts and runs the task scheduler only if the `sfti_Environ_g.Global` is equal to 1. `sfti_Environ_g.Global` is not decremented until servicing HISRs and scheduling is complete. This convention prevents conflicts between the system calls that service software interrupts and run the task scheduler after a critical region, and the LISR dispatcher, which also services pending software interrupts.

3.1.3 Memory Manager

A memory segment is a large contiguous block of memory used for dynamic data and stacks. Each memory segment must have its own set of low-level memory allocation services, which is called a memory manager. Segments usually have individual qualities, such as a size and access speed.

CORTEX offers two types of memory segments, static and dynamic.

Static segments are created at system initialization time by the `segm_Init()` routine. To specify required memory layout, an application must define the `segm_Table_g table`. Static memory segments can be linked together. When a memory block with segment linkage is allocated and the required amount of memory is not available, the following linked memory segment is used. This technique is also called overlapping.

Dynamic memory segments can be created by an application at runtime. A memory block for a dynamic segment can be allocated from a static segment, from another dynamic segment, or defined statically as an array.

At least one memory segment must be defined as a system segment. A typical configuration has one system segment and one default memory segment. The system segment is used for allocating all the memory used by the kernel (for example, a task’s control blocks) while the default segment is used for such functions as stack allocation.

The static segments are defined in the *kernel/src/seg_data.c* file. In this file, the `segm_Table_g` table is filled cell by cell. By default, the first segment is the system segment, the second is the default segment and the third is the fast segment. The user can reorder these segments and add additional segments. Symbols used to define the static segments include

- SRAM_BASE—the start address of normal static segments
- SRAM_LENGTH—the length of the memory block associated with normal memory segments
- CRAM_BASE—the start address of fast static segments
- CRAM_LENGTH—the length of the memory block associated with fast memory segments

The user must define these variables in the linker configuration file *bsp/sc100/link.cmd*.

Each segment has a memory manager assigned to it. A memory manager consists of a collection of functions that allocate, deallocate, and reallocate memory. Some memory managers can run faster than others, but incur trade-offs with other aspects of system performance. CORTEX offers three types of memory management and allows the use of custom-defined managers.

3.1.4 System Clock

The system clock provides a set of services which mark the time flow in the kernel. System time is expressed as a number of system ticks, where one system tick equals the time interval between two consecutive interrupts from the system clock.

The system timer manager counts the number of system ticks (as well as the number of seconds and nanoseconds) since system startup. The system tick counter eventually wraps around and begins to count from 0. Tick counter reset does not affect timeout processing.

3.2 HAL—Platform-Dependent Layer

In order to port CORTEX to a new platform, several hardware-dependent functions in the hardware abstraction layer (HAL) must be written. This section lists these functions according to the file names in which they appear.

Code Listing 1. Functions in *ports/sc100/src/sc100.c*.

```
/* return highest interrupt priority level */
hrdi_Priority_t hrdi_MaxPriority(hrdi_Priority_t Prio_1_a, hrdi_Priority_t Prio_2_a);

/* Do some additional CPU setup to enable specified interrupt source. This service is
called from hrdi_Install(). */
crtx_Void_t hrdi_EnableVector(hrdi_ISRCB_t * pISRCB_a);

/* Do some additional CPU setup to disable specified interrupt source. This service is
called from hrdi_Uninstall(). */
crtx_Void_t hrdi_DisableVector(hrdi_ISRCB_t * pISRCB_a);

/* Calculate interrupt mask for specified interrupt vector. For HRDI_PRIO_LEVEL_MODEL
this service returns interrupt priority mask (IPM) which is calculated from priority
level assigned to specified interrupt source. */
hrdi_Mask_t hrdi_Mask(hrdi_Vector_t Vector_a);

/* Initialize LISR stack frame on LISR registration */
crtx_Void_t hrdi_MakeLisrStackFrame(hrdi_ISRCB_t * pLisr_a);

/* Prepare LISR stack immediately before switching to LISR */
crtx_Void_t hrdi_PrepareStack(hrdi_ISRCB_t * pLisr_a);

/* Check LISR stack frame. */
crtx_Bool_t hrdi_CheckStack(hrdi_ISRCB_t * pLisr_a);

/* LISR shell. Prepares and activates LISR. Called by dispatcher */
crtx_Void_t hrdi_Shell(hrdi_Vector_t Vector_a);

/* Register direct ISR */
crtx_Void_t hrdi_RegisterDISR(hrdi_ISRCB_t * pDisr_a);

/* Unregister direct ISR */
crtx_Void_t hrdi_UnregisterDISR(hrdi_ISRCB_t * pDisr_a);

/* Register LISR dispatcher */
crtx_Void_t hrdi_RegisterDispatcher(hrdi_ISRCB_t * pLisr_a);

/* Unregister LISR dispatcher */
```

```
crtx_Void_t hrdi_UnregisterDispatcher(hrdi_ISRCB_t * pLisr_a);

/* Initialize HISR stack frame on HISR registration. */
crtx_Void_t sfti_MakeHisrStackFrame(sfti_ISRCB_t * pHisr_a);

/* Prepare HISR-stack immediately before switching to HISR */
crtx_Void_t sfti_PrepareStack(sfti_ISRCB_t * pHisr_a);

/* Check HISR stack frame. */
crtx_Bool_t sfti_CheckStack(sfti_ISRCB_t * pHisr_a);

/* Initialize thread's stack frame */
crtx_Void_t thrd_MakeThreadStackFrame(thrd_TCB_t * pTCB_a);

/* Check thread's stack */
crtx_Bool_t thrd_CheckStack(thrd_TCB_t * pTCB_a);

/* Compares two stack pointers and returns one closest to the top */
crtx_Void_t * thrd_MaxSP(crtx_Void_t * pSP1_a, crtx_Void_t * pSP2_a);

/* Returns adjusted value of max SP. Must be implemented if
THRD_THREAD_TRACKER_ENABLED is defined and has a value of 1*/
crtx_Void_t * thrd_AdjustMaxSP(thrd_TCB_t * pTCB_a);

/* Calculate thread's stack usage in % */
crtx_Int_t thrd_StackUsage(crtx_Uint32_t * pStackTop_a,crtx_Uint32_t * pStackBase_a,
crtx_Void_t * pSP_a);

/* Returns list of thread's stack frames */
crtx_Int_t thrd_GetStackFrames(thrd_TCB_t * pTCB_a, crtx_Int_t Size_a,
thrd_StackFrame_t * pFrames_a);

/* Default idle thread */
crtx_Void_t thrd_IdleThread(crtx_Void_t);

/* System Ticks Manager low-level Interrupt Service routine */
crtx_Void_t tick_LISR(hrdi_Vector_t Vector_a, hrdi_Context_t Context_a);

/* Default routine to setup system timer */
crtx_Void_t tick_SetupSystemTimer(crtx_Void_t * pContext_a)

/* Returns number timer's clocks expired since last timer reset */
crtx_Uint32_t tick_ClocksSinceReset(crtx_Void_t)

/* SC100 port initialisation routine - called from init_System() */
crtx_Void_t port_Init(crtx_Void_t)

/* Outputs error message to serial port and stops all execution. Called from
syst_Fatal() */
crtx_Void_t port_Fatal(crtx_Void_t);

/* Outputs error message to serial port and stops all execution. Called from
syst_Abort(). */
crtx_Void_t port_Abort(crtx_Void_t);

/* Stops the system (on normal exit). Called from syst_Exit() */
crtx_Void_t port_Exit(crtx_Void_t);

/* Prepare serial port to output some data */
crtx_Void_t port_InitSerial(crtx_Void_t);

/* Output data byte to serial port when system has been crashed */
crtx_Void_t port_Putc(crtx_Char_t Char_a);


/* Set of C-style allocate/deallocate memory functions */
```

```
crtx_Void_t * malloc(size_t Size_a);
crtx_Void_t free(crtx_Void_t * Addr_a);
crtx_Void_t * calloc(size_t Nelem_a, size_t Size_a);
crtx_Void_t * realloc(crtx_Void_t * OldAddr_a, size_t Size_a);
```

Code Listing 2. Functions in *ports/sc100/src/hwi_asm.asm*.

```
/* Activate interrupt subsystem. */
crtx_Void_t hrdi_Start(crtx_Void_t);

/* Increment specified location as an atomic operation and return new value */
crtx_Int_thrdi_Inc(crtx_Int_t * pVar_a);

/* Decrement specified location as an atomic operation and return new value */
crtx_Int_thrdi_Dec(crtx_Int_t * pVar_a);

/* Set specified location to zero and return previous value */
crtx_Int_thrdi_Zero(crtx_Int_t * pVar_a);

/* Assign new value to specified location and return previous value
*/
crtx_Int_thrdi_Assign(crtx_Int_t * pVar_a, crtx_Int_t Value_a);

/* Logically AND specified location and mask as an atomic operation and return new
value */
hrdi_Mask_thrdi_And(hrdi_Mask_t * pVar_a, hrdi_Mask_t Mask_a);

/* Logically OR specified location and mask as an atomic operation and return new
value */
hrdi_Mask_thrdi_Or(hrdi_Mask_t * pVar_a, hrdi_Mask_t Mask_a);

/* Logically XOR specified location and mask as an atomic operation and return new
value */
hrdi_Mask_thrdi_Xor(hrdi_Mask_t * pVar_a, hrdi_Mask_t Mask_a);

/* Invert all bits at specified location as an atomic operation and return new value
*/
hrdi_Mask_thrdi_Not(hrdi_Mask_t * pVar_a);

/* Disable interrupt sources according interrupt mask. Returns mask of interrupt
sources disabled by this service */
hrdi_Mask_t hrdi_Disable(hrdi_Mask_t Mask_a);

/* Enable interrupt sources specified by argument */
crtx_Void_t hrdi_Enable(hrdi_Mask_t Mask_a);

/* Convenient and efficient way to disable registerd hardware interrupts for a short
time. */
hrdi_FastIntrCookie_t hrdi_FastIntrDisable(crtx_Void_t);

/* Re-enable interrupts disabled by hrdi_FastIntrDisable(). */
crtx_Void_t hrdi_FastIntrEnable(hrdi_FastIntrCookie_t Cookie_a);

/* Quickest way to disable global interrupt flag (whatever it is on current CPU. CPU
specific implementation. */
hrdi_GlobalIntrCookie_thrdi_GlobalIntrDisable(crtx_Void_t);

/* Re-enable interrupts disabled by hrdi_GlobaIntrDisable(). */
crtx_Void_t hrdi_GlobalIntrEnable(hrdi_GlobalIntrCookie_t Cookie_a);

/* Checks for nested hardware interrupts. */
crtx_Boolean_t hrdi_IsNested(crtx_Void_t);
```

```
/* Set LISR interrupt mask. This service is only alowed from hrdi_Shell() to allow
proper nested interrupt detection for some ports. */
crtx_Void_t hrdi_SetLisrIntrMask(crtx_Mask_t Mask_a);

/* Set new CPU priority level and returns previous CPU priority level. */
hrdi_Priority_t hrdi_SetPrioLevel(hrdi_Priority_t);

/* Return number of the most significant bit which is set. */
crtx_Int_t hrdi_GetMsbMask(crtx_Mask_t Mask_a);

/* Switch to LISR stack and execute LISR handler. */
crtx_Void_t hrdi_SwitchStack(crtx_Void_t * pNewSP);

/* Restore old stack. This procedure is never ever called directly and always
activated as result of steping through LISR stack frame. */
crtx_Void_t hrdi_RestoreStack(crtx_Void_t);
```

Code Listing 3. Functions in *ports/sc100/src/swi_asm.asm*.

```
/* Switch to HISR stack and execute HISR handler */
crtx_Void_t sfti_SwitchStack(crtx_Void_t * pNewSP);

/* Recover HISR stack. Only called from stack frame. */
crtx_Void_t sfti_RestoreStack(crtx_Void_t);
```

Code Listing 4. Functions in *ports/sc100/src/thr_asm.asm*.

```
/* Get caller's stack frame pointer (may return NULL if service is not available for
particular port) */
crtx_Void_t * thrd_GetStackFP(crtx_Void_t);

/* Switch thread context. Called from thrd_Scheduler() and
thrd_StartMultiThreadEnviron() */
crtx_Void_t thrd_SwitchStack(crtx_Void_t ** ppCurrSP, crtx_Void_t ** ppNextSP);

/* Pop stack arguments into CPU registers. */
crtx_Void_t thrd_ArgsToRegs(crtx_Void_t);
```

Code Listing 5. Functions in *ports/sc100/src/hwi_disp.asm*.

```
/* Hardware interrupts dispatcher */
crtx_Void_t hrdi_Dispatcher(crtx_Void_t);
```

Code Listing 6. Functions in *exbsp/sc100/src/plt_init.c*.

```
/* Does all platform specific initialization. */
crtx_Void_t pltf_Init(crtx_Int_t ArgC_a, crtx_Void_t * pArgV_a, crtx_Void_t *
pEnvV_a);
```

4 Porting Details

This section details the porting of CORTEX-based code to the MSC8101 platform, including platform initialization, inerrupt management, task management, system clock, memory management, and system build.

4.1 Platform Initialization

4.1.1 Boot Code

CORTEX provides a default bootstrap code that is used if the `ENVI_BOOT_USE_CORTEX_BOOTSTRAP_CODE` parameter is set to 1. Program execution begins with the first entry in the interrupt table (also used for TRAP exception). The default bootstrap code is located in the file *ports/sc100/src/boot.asm*. The code copies data from ROM to RAM if necessary, assigns initial values to registers (especially SR and SP), sets the parameters (`ArgC_a` and `ppArgV_a`) and the environment (ppEnvV_a), and then calls the `main()` routine. This file also contains the interrupt table and certain functions used by the debugger for input/output operations.

4.1.2 Linker Command File

The executable produced by the linker is not relocatable, which means that the addresses of each program section must be known at link time. This information is included in the linker command file *bsp/sc100/src/link.cmd*. In addition to section addresses, this file also contains definitions of symbols required by CORTEX, such as the primary/secondary interrupt table boundaries and memory for CORTEX static segments.

4.2 Interrupt Management

4.2.1 The Interrupt Table

The MSC8101 has two interrupt tables, one for the Programmable Interrupt Controller and one for the SIU-CPM interrupt controller. The PIC and SIC interrupt tables, each have 64 entries. Because CORTEX assumes that all interrupts are in the same table, the two tables are concatenated into a single 128-entry table. The first 64 entries are accessed normally via the PIC. The last 64 entries correspond to SIC interrupts and are accessed through a special interrupt handler placed in the 48^{th} entry of the table (which corresponds to SIC interrupt requests). This handler reads the interrupt offset from the SIVEC register and jumps to the appropriate entry in the second half of the table. In order to register an interrupt handler for the n^{th} SIC interrupt, `hrdi_Install()` must be called with n+64 as an argument. Figure 2 illustrates the structure of the interrupt table.

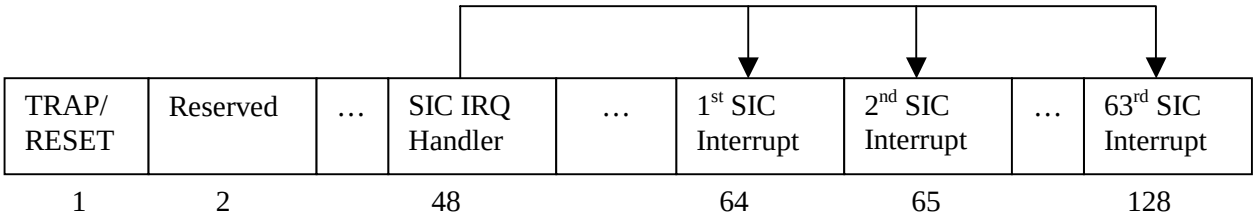

Figure 2. MSC8101 Interrupt Table

Each entry in the interrupt table is 64 bytes wide. The table occupies $2000 bytes of memory (8 KB). To register an interrupt handling routine, a standard sequence must be written in the corresponding interrupt table entry. The sequence for a LISR and DISR are shown in Code Example 7 and Code Example 8 respectively.

Code Example 7. Code sequence for serving a LISR.

```
DI
JSR Dispatcher
RTE
```

Code Example 8. Code sequence for serving a DISR.

```
DI
JMP DISR
```

To unregister a DISR or LISR the corresponding interrupt table is registered as unhandled LISR, called through the default dispatcher.

4.2.2 Default LISR Dispatcher

The default LISR dispatcher is a key component of the hardware interrupt manager. It can be used to call all LISRs or as a model to write a custom dispatcher. The main operations performed by the dispatcher include:

- Saving the context of the interrupted task. All core registers must be saved here. It is not possible to reduce overhead by using a dispatcher that saves only a subset of the registers used in the LISR, because every dispatcher must service software interrupts, which can lead to task switching.
- Incrementing the `hrdi_NestedPtr_g` variable and enabling the hardware global interrupt flag. The global interrupt flag must stay disabled until `hrdi_NestedPtr_g` is incremented.
- Calculating an interrupt vector number. This operation uses the return address from the dispatcher, which points to the interrupt table entry from which the dispatcher was called. It is for this reason that a dispatcher must be called with a JSR rather than a JMP instruction. The desired address is obtained by subtracting the start address of the interrupt table from the return address and dividing the result by 64, the size of an interrupt table entry.
- Activating a registered LISR (providing LISR parameters and switch stacks if required). This is done by calling the `hrdi_Shell()` function.
- Restoring the previous interrupt mask and servicing all pending HISRs if there are no nested interrupts.

- Decrementing the `hrdi_NestedPtr_g` variable, restoring the context of the interrupted task and returning to that task. This must be in a critical region for a dispatcher that has just serviced software interrupts, otherwise an unlimited number of stack frames on the stack can accumulate.

The `hrdi_NestedPtr_g` global variable stores a pointer to a counter that holds the level of interrupt nesting. Only nested LISRs are counted here. DISRs should never modify this variable. The dispatcher needs to know the level of interrupt nesting because servicing software interrupts can only be done in the outermost dispatcher.

4.2.2.1 Servicing Software Interrupts

Any software interrupts triggered by LISRs must be serviced after all LISRs have completed. The dispatcher must perform this task as the last operation before restoring the program context if and only if the interrupt nesting level is equal to one. Servicing software interrupts is done by calling the `hrdi_ServicePending()` and `hrdi_CheckPending()` functions. The scheduler is called after all software interrupts have been serviced. This means that any task switching occurs inside the dispatcher. When the current task is rescheduled the dispatcher ensures the restoration of the task context. This process raises two issues:

- The previous interrupt priority level must be restored before servicing software interrupts; otherwise, if a task switch occurs, some interrupts may be disabled for an indefinite amount of time until the processor returns to the current task. Restoring the interrupt priority level is done by reading the previous value of the status register (SR) from the stack and writing the Interrupt Priority Level (IPL) bits in the SR with their previous value.
- Under no circumstances can the dispatcher be interrupted by another LISR before `hrdi_NestedPtr_g` is incremented. If this were to happen, the interrupting dispatcher would service software interrupts (because `hrdi_NestedPtr_g` would be 1), and very likely switch tasks, which could indefinitely postpone servicing the interrupted LISR. For this reason the first instruction executed after the occurrence of the interrupt, even before the dispatcher is called, is DI. The dispatcher must execute an EI instruction after the `hrdi_NestedPtr_g` variable is incremented. The SIC interrupt handler also starts with a DI instruction. The DI must be executed immediately after the occurrence of the interrupt because only the first three execution sets are guaranteed to enter execution before another interrupt request can be serviced. The EI should also be executed as soon as possible, to minimize response time for higher priority interrupts (the default dispatcher does this in the context saving zone, after sufficient registers have been pushed on the stack).

In addition, the dispatcher should not be interrupted after `hrdi_NestedPtr_g` is decremented to zero, otherwise an unlimited number of stack frames can accumulate. (The interrupting dispatcher could service software interrupts, then after decrementing hrdi_NestedPtr_g could also be interrupted, and so on.)

The second issue above also affects DISRs— a DISR should not be interrupted by a LISR. Since DISRs do not use the `hrdi_NestedPtr_g` variable, the dispatcher has no way of knowing that it has interrupted a DISR, so it will service HISRs.

LISRs cannot be used for NMIs, because they are not affected by the DI flag, which is used by the dispatcher to implement critical regions and by DISRs to protect themselves from LISRs. An NMI can interrupt a DISR or another dispatcher before it increments `hrdi_NestedPtr_g` and then service HISRs.

The maximum number of interrupt stack frames is 9—one for the dispatcher that was interrupted while servicing HISRs, plus 8 others for the worst case of interrupt nesting (one for every valid IRQ priority level and one for NMI). StarCore internal exceptions can affect this calculation.

4.2.3 The LISR and HISR Stacks

An extra amount of space must be allocated on the stack for each thread to accommodate worst-case interrupt nesting. This is a waste of memory, because the extra space will never be entirely used. Interrupts occur in only one thread at a time, but there is no way of knowing which thread it will be, so the required space must be allocated on all stacks.

To solve this problem, StarCore provides a special exception stack used by interrupt routines and system services. Two stack pointers are provided, NSP (Normal Stack Pointer register) and ESP (Exception Stack Pointer register). Extra stack space is allocated only once, on the exception stack. Each time an interrupt occurs the ESP register is activated and all stack operations use the exception stack. OS functions are then called via the TRAP instruction. This mechanism is based on the assumption that the exception stack is empty when a task switch occurs. This is not the situation in CORTEX. As described in Section 4.2.2.1 on page 15, task switching caused by an interrupt takes place inside the LISR dispatcher. As a result, context saving and servicing HIRSs must be done on the task's stack. Only nested interrupts or DISRs can use the special stack. System functions should also use the task's stack, since task switching occurs inside them. Therefore StarCore's stack switching system must be ignored by using only the ESP register and never the NSP.

CORTEX provides its own mechanism to alleviate this problem—private stacks for LISRs and HISRs. When an LISR occurs, the dispatcher still uses the task's stack, but a stack switch is performed before calling the LISR. The LISR and all nested interrupts use the LISR private stack. Thus, stacks do not require extra space for nested interrupts. LISRs with the same priority level and HISRs can use the same stack, since they cannot interrupt each other.

4.2.3.1 Switching Stacks

If an LISR or HISR has a private stack, its address is stored in the ISR control block. Figure 3 shows how the LISR stack is set up just prior to a stack switch. (The HISR stack is similar.)

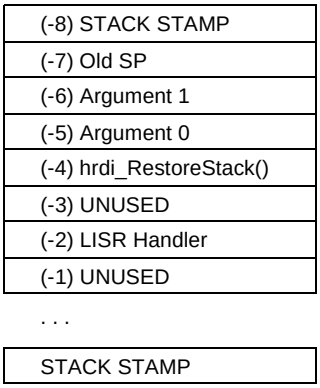

Figure 3. LISR Stack Frame

The stack stamps are used to check for stack overflow or underflow during debugging. The old SP is stored on the stack so it can be restored after LISR has finished. The function that actually performs the stack switch, `hrdi_SwitchStack()`, is then called. After setting the value of SP as shown in Figure 3, the new stack is active. The two arguments are then loaded into the appropriate registers, according to ABI conventions for calling C functions. The return from `hrdi_SwitchStack()` starts the LISR handler (the LISR handler address is considered the return address). After returning from the LISR handler, `hrdi_RestoreStack()` restores the SP to its previous value.

CORTEX assumes that stacks are aligned on 4-byte boundaries, and all kernel functions that allocate stack space also make this assumption. Because the StarCore stack must be aligned on 8-byte boundaries, the function that prepares the LISR stack must adjust the received memory block to ensure that the stack start and end are properly aligned. The application programmer should be aware that the available stack space may be a bit less than required.

4.2.4 Activating/Deactivating Interrupts

CORTEX provides the application with a set of functions for enabling/disabling hardware interrupts to allow implementation of critical regions. Some of these functions are wrapped in hardware-independent functions, which allows nesting of enable/disable pairs. These functions include

- `hrdi_SetPrioLevel()`—sets the interrupt priority level to the value received as a parameter. The old priority level is returned. This function changes the IPL bits in the SR and returns their previous value shifted right to obtain a number between 0 and 7.
- `hrdi_GlobalIntrDisable()/hrdi_GlobalIntrEnable()`—disables/enables all interrupts by setting the IPL to 7. CORTEX documentation suggests that `hrdi_GlobalIntrDisable()` and `hrdi_GlobalIntrEnable()` should use the global interrupt disable bit, DI. This cannot be done because there are regions guarded by `hrdi_GlobalIntrDisable()` and `hrdi_GlobalIntrEnable()` in some kernel functions that contain atomic operations which also use the DI bit. `hrdi_GlobalIntrDisable()` returns a cookie representing the previous value of the IPL bits, not right-shifted, which must be passed to `hrdi_GlobalIntrEnable()` to restore the previous priority level.
- `hrdi_FastIntrDisable()/hrdi_FastIntrEnable()`—quickly disables all interrupts. These functions set/reset the DI bit in the SR, which is the fastest way of disabling/enabling interrupts. Regions guarded by these functions should not contain atomic operations or other functions that use the DI bit.
- `hrdi_Disable()/hrdi_Enable()`—these functions disable/enable interrupt sources specified by the mask received as a parameter. Disabling interrupts is done by setting the priority level to the highest priority interrupt specified in the mask. `hrdi_Disable()` returns the mask of interrupts disabled by this function, which should be used by `hrdi_Enable()` to reenable interrupts.

Masks of interrupts handled by `hrdi_Disable()` and `hrdi_Enable()` contain one bit for each priority level, where the rank 0 bit corresponds to priority level 1. The result of `hrdi_Disable()` is

- 0 if all interrupts in the mask are already disabled
- $(2^{newIPL} – 2^{oldIPL})$ otherwise

`hrdiEnable()` lowers the priority level to the highest possible level that still enables all interrupts specified in mask. `hrdiEnable(0)` has no effect. `hrdi_Disable()` never lowers the IPL, and `hrdi_Enable()` never raises it.

The function `hrdi_Mask()` can be used to obtain the priority mask of an interrupt with a given vector number. The priority level of the interrupt is obtained from the table of registered LISRs.

4.2.5 Atomic Operations

The hardware interrupt manager features a set of atomic operations. These are simple read-modify-write operations that are guaranteed to be executed without interruption, such as increment, decrement, logical or, etc. The implementation of these functions ensures atomicity by enabling/disabling interrupts. The pattern followed by each of these functions is shown in Code Example 9.

Code Example 9. Pattern for Atomic Functions

```
DI   read

modify

EI   write
```

4.3 Task Management

The entire life cycle of a task is laid out on the stack, including all of its functions and their arguments. When a function is completed, the next function in the task's life cycle is called with a simple RETURN instruction. This stack model is based on the fact that a RETURN instruction pops the new value of the program counter (PC) from the stack. This mechanism simplifies things considerably because there is no need to remember the exact point in the life cycle when the task is interrupted, and the task switch is resumed only to the stack switch. Figure 4 illustrates the stack at the beginning of a task life cycle. (The stack is modified during the task's lifetime).

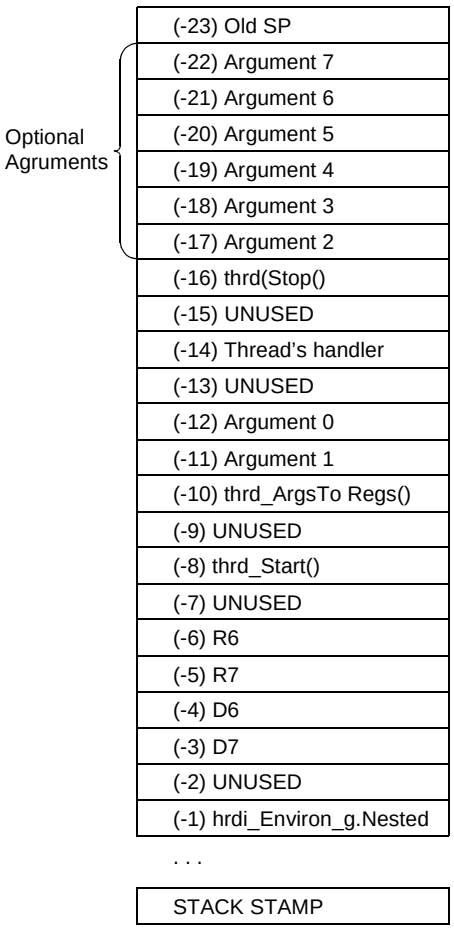

Figure 4. Task Stack Frame

When a task switch occurs, the `thrd_SwitchStack()` function executes. This function performs the following tasks:

- Saves the R6, R7, D6, D7 registers (ABI conventions) and the `hrdi_Environ_g`.Nested value of the current task on its stack
- Saves the old SP in a special field in the preempted task control block
- Loads the stack pointer value of the new task in the SP register
- Loads the values from the next task's stack in the R6, R7, D6, D7 registers and the `hrdi_Environ_g`.Nested field.

After the `thrd_SwitchStack()` executes, a RETURN instruction is executed (the new task stack is now active). In the case of a running task that has regained the processor, the RETURN instruction starts the previously-interrupted thread-handler function.

When a new task is created and gains the processor for the first time, the RETURN instruction that executes after the `thrd_SwitchStack()` function loads the address of the `thrd_Start()` function in the PC. When the `thrd_Start()` function finishes, the RETURN instruction loads the address of the `thrd_ArgsToRegs()` function in the PC register. This function is necessary because the ABI convention requires that every function receives its first two arguments in the D0/R0 and D1/R1 registers. The `thrd_ArgsToRegs()` function POPs Argument 0 and Argument 1 from the stack and loads them in the appropriate registers. When this function finishes, the RETURN instruction loads the address of the thread's handler function in the PC register. This function describes what the task actually does. When this function finishes, the task is completed, and the next RETURN instruction loads the address of the `thrd_Stop()` function in the PC register. This function tells the kernel that the task is finished and frees all resources occupied by the task.

4.3.1 Task's Stack

The stack for a task must be properly configured before the task can be created. As explained previously, the functions that describe the task's life cycle are placed on the task's stack. When one function finishes, the next function is called with the RETURN instruction, which loads the address found at SP–1 in the PC register. The correct start address for each function must be placed on the stack so that the function can be called properly. There are several steps involved in creating a stack properly:

1. The stack must be correctly allocated and aligned on 8-byte boundaries.
2. All the cells in the stack are initialized with a predefined pattern.
3. Determine how many arguments this task handler will receive. If the argument the number is even, a 4-byte section from stack must remain unused in order to comply with the stack alignment requests.
4. Place all of the task's arguments on the stack except the first two, which will be placed later.
5. Place the `thrd_Stop()` and `thrd_Entry()` (the task's handler) functions addresses on the stack along with the necessary unused gaps for stack alignment.
6. Place the first two arguments of the task on the stack., followed by the address of the `thrd_ArgsToRegs()` function. This function loads the two arguments into the appropriate registers in compliance with the ABI convention.
7. Place the address of the `thrd_Start()` function on the stack.
8. Save the R6, R7, D6, D7 registers on the stack to comply with ABI convention.
9. Put the `hrdi_Environ_g.Nested` field on the stack. This field contains the LISR's nesting level. The HISRs will not be serviced unless this field is 0 or 1.

4.3.2 Stack Frame Tracing

Suppose that function 1 calls function 2 which calls function 3 … which calls function n. Stack frame tracing means that function n can access the stack of any of functions 1 to n–1. According to the ABI convention, if stack frame tracing is necessary, the calling function must save the start address of its stack in the R7 register before it calls another function. The called function in turn must save whatever it receives in R7 on the top of its stack. This effectively creates a sort of a chained list of function stacks, making it easy to access any particular stack. However, this mechanism is merely an ABI suggestion. The Metrowerks Enterprise C compiler does not currently implement this convention, so stack frame tracing was not implemented in this project.

4.3.3 The Idle Task

This task has the lowest possible priority. Its function is to assure that the processor always has a task to run (otherwise the system will finish its execution). All this task does is wait for an event to occur. The code for this task is shown in Code Listing 10.

Code Listing 10. Idle Task

```
for( ; ; )
{
    wait();
}
```

4.4 System Clock

The system clock component uses the Periodic Interrupt Timer (PIT) as a clock source. The PIT receives its input clock from the Baud Rate Generator 1 (BRG1). Two main functions were written for this component—the clock initialization function and the clock LISR.

4.4.1 Clock Initialization

Clock initialization consists of three operations:

1. Configure the BRG1 to generate an 8192 Hz signal to PIT.

BRG period is:

$$\text{BRGCLKOUT} = \frac{2 \times \text{Fcpm}}{\text{BRG_DF} \times \text{PRESCALE} \times \text{Divider}} \qquad \textbf{Eqn. 1}$$

where:

- the post-divider BRG_DF is set by the SCCR register and can be 4, 16 (default) 64 or 256
- PRESCALE: set by the BRGC1 register (CD bits) and can have values in a range from 1 to 4096
- Divider: set by the BRGC1 register (DIV16 bit) and can be 1 or 16

2. Activate PIT and set timeout period. The timeout period is set according to the ENVI_TICK_SYSTEM_TICKS_PER_SEC environment parameter
3. Enable PIT interrupt requests. Since the PIT belongs to the SIU, the PIT interrupt request must be enabled in the SIC (SIMR_H register, bit 1), and the SIC interrupt request must be enabled in the PIC (ELIRE register, bits 3:0). Note that setting a priority level for the SIC interrupt request in the PIC will affect all other SIC interrupts.

4.4.2 Clock LISR

The LISR performs the following set of operations required by CORTEX:

- Adjust the system time
- Run the application LISR handler
- Increment a special counter to be read by the HISR and trigger clock HISR. (This component is not described here because it is hardware-independent.)

In addition, the PIT interrupt must be acknowledged by resetting the PS bit in the PISCR register (at the PIT level) and bit 1 in the SINPR_H register (at the SIC level).

4.5 Memory Management

The processor-dependent part of the memory manager includes four functions: `malloc()`, `calloc()`, `free()` and `realloc()`. These functions are not really processor dependent, but CORTEX enables custom configuration of the functions. The solution chosen here was to use some low-level functions provided by CORTEX for memory management.

- `malloc()`—this function uses the `dmem_Alloc()` function which receives the following parameters:
  - — The memory segment where the memory will be allocated (in our case this segment is the default segment)
  - — The size of the memory block to be allocated plus a cell which will contain the actual block size (this information is needed by the `free()` function)
  - — The block alignment (which in our case is 4 bytes).

  The function receives a pointer to the allocated block from the `dmem_Alloc()` function. If the pointer is not NULL, then its size is placed at the top of the block. The function returns a pointer to the actual memory block.
- `calloc()`—same as `malloc()` except that `dmem_Calloc()` rather than `dmem_Alloc()`is used, and the pointer to the actual memory block that is returned is initialized with zeros.
- `free()`—this function receives an address and frees a memory block from that address but only after it verifies that the block address is correct.
- `realloc()`—this function reallocates a memory block from an old address to a new address using the `dmem_Realloc()` function. The `dmem_Realloc()` function receives the old address and the size of the new memory block and returns the address of the newly allocated block.

4.6 Building the System

Two approaches were used in order to build the system. The first one was to use the makefiles system proposed by CORTEX and the second one was to use CodeWarrior projects.

4.6.1 Makefiles

The configuration files are designed to provide a generic and consistent way to handle different development tool chains and target platforms. Most of the make rules are defined to be tool-independent and require minimal input from tool-specific configuration files. This approach also facilitates extending a set of supported development tools without modification of the existing configuration file.

The main configuration files provided by CORTEX include:

- *gmake/template.cf*—The template configuration file directly or indirectly includes all other configuration files in order to produce the final makefile which contains all rules and definitions required to assemble, compile and link the kernel, its components and final applications, generate dependencies, install pre-compiled libraries, etc.
- *gmake/rules.cf*—The rules configuration file contains definitions for all tool independent-rules. It avoids duplication among configuration files for different tool chains and maintains a consistent set of supported rules. The rules rely on additional macro definitions provided by tool-specific configuration files (in our case the *gmake/sc100.scc* file).
- *gmake/params.cf*—The parameter file contains definitions common to all development environments.
- *gmake/tool.tb*—This file functions as a switch between all tool-specific configuration files. It conditionally includes all other tool-specific configuration files and guarantees that only one of them is included in the final product. This file must be updated when configuration files for a new development tool are implemented. The TOOL macro is used to specify the required development tool.
- *gmake/unknown.tl*—This configuration file is used for unknown tool chains. *gmake/tool.tb* includes this file if the TOOL macro specifies an unknown development tool and contains only a basic set of definitions.
- *gmake/bsp*—This directory contains configuration files for all supported target systems (board support configuration files). The name of the file must match the abbreviated name of the target platform (specified via the TARGET parameter).

For a new CORTEX port, several steps must be followed. First, in the *gmake/tool.tb* file some new lines must be added to include the file that specifies the compiling rules with the new tools. The lines added to *gmake/tool.tb* for our project are shown in Code Listing 11.

Code Listing 11. Updating *gmake/tool.tb*

```
# ENTERPRISE tools for SC100 architecture

ifeq (${TOOL},scc100)
include sc100.scc
endif
```

Next, the *gmake/sc100.scc* configuration file must be written. This is the only configuration file that must be written by the programmer. To facilitate this process, CORTEX provides an example file named *gmake/common.gcc* which contains the compiling, assembling and linking rules common to all GCC C/C++ compilers. Once a particular C compiler is chosen (in our case the StarCore compiler) one simply modifies some symbols definitions and renames the file with the platform name. (We used the Metrowerks Enterprise C compiler.) The *gmake/sc100.scc* file contains the following items:

- Paths to the new compiler and to the libraries it uses
- A list of file extensions recognized by the new compiler
- The compiling options for many kinds of input files. Our project used C files, assembler files. We also used an archive tool called *sc100-ar.exe* that receives object files and converts them to libraries.

Finally, the macros used by the *gmake/rules.cf* file to compile the source files must be defined. Our project had rules for:

- compiling C-files
- compiling ASM files

- linking object files
- linking object files with debugging support
- creating libraries from object files

For example, for assembling the ASM files we have a rule in *gmake/rules.cf* file shown in Code Example 12.

Code Example 12. Pattern Rule for Assembling ASM Files

```
# Pattern rule to assemble assembler's sources to object files

${OBJ_DIR}/%.${OBJ_EXT}: ${SRC_DIR}/%.${AS_EXT}
@${COMPILE_ASM}
```

In order to make this rule work, the COMPILE_ASM macro must be defined. This is done in the *sc100.scc* file, as shown in Code Example 13.

Code Example 13. Assembling Rule

```
# Rule to assemble

define COMPILE_ASM
${ECHO} "Assemble  from $<"
${ECHO} "          to   $@"
(${AS} ${AS_OPTS} ${ASM_ENVI_DEFINES} ${AsmOutputFileName} $@ -- $<)
test -h ${DBG_DIR}/${<F} || ${LN} -fs $< ${DBG_DIR}/${<F}
endef
```

The assembling line will be translated as shown in Code Example 14.

Code Example 14. Final Assembling Command

```
asmsc100 –q -s all -o elf –I "include paths" –b input_file.asm -- output_file.eln
```

4.6.2 CodeWarrior Project

This project followed the same pattern as the makefiles. A project was created to build the system. A target was created for every section of CORTEX, resulting in four different libraries—kernel, sc100, excore, and exbsp. The libraries representing the OS were linked with the application in a new project and the resulting binary file was downloaded on the MSC8101 ADS.

Figure 5 is a snapshot from CodeWarrior showing the projects for one of the test applications listed in Section 5 on page 24. The right-hand window contains the sources for CORTEX and the left-hand window contains the application and the four libraries.

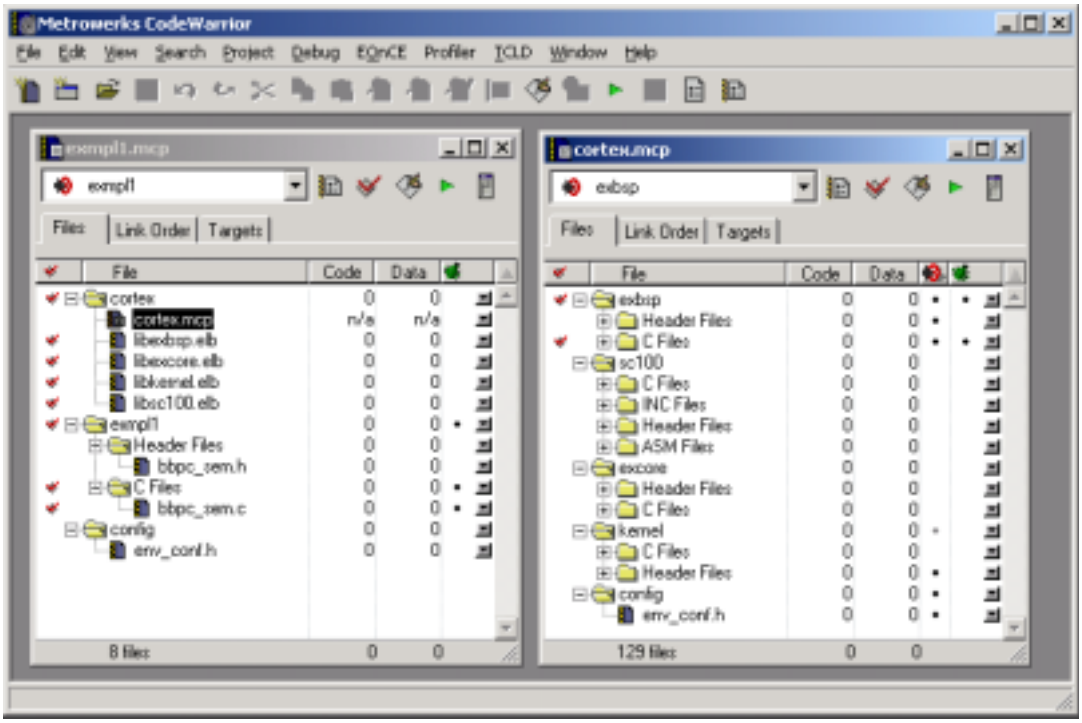

Figure 5. CodeWarrior Project for the 'Producer-Consumer Problem' Test Application

5 Testing

CORTEX was tested on hardware using the Metrowerks CodeWarrior IDE. The CodeWarrior IDE provides a complete environment for developing and debugging StarCore applications. The compiler produces an executable file which is downloaded to the board through the parallel port using a Command Converter Server (CCS).

CodeWarrior offers real-time debugging facilities, enabling the user to observe the processor's registers and memory at any point in the program execution. A `printf()` function is also available which prints messages on the computer screen.

To test the port, some classic operating system algorithms were used, including

- the producer-consumer problem with a bounded buffer
- the philosophers problem
- implementation of semaphores using mutexes

6 Results

The performance figures presented in Table 1 were measured using the CodeWarrior simulator.

Table 1. Some Performance Figures

Function	Speed (cycles)
Task switch	1259
Create task	391
thrd_Scheduler()	384
Create software interrupt	377
malloc()	360
calloc()	453
realloc()	1547
free()	363
Context save	33
Boot code	48
Default LISR dispatcher	88
Create LISR	309
Clock LISR	50
Clock HISR	167

7 Conclusions

The MSC8101 is a very powerful device with several peripherals and a high-performance DSP core capable of running complex applications. Writing such applications without the assistance of an RTOS can be very cumbersome; tasks such as scheduling, synchronization and memory management can become so cumbersome that the original purpose of the application can be lost, and development time can increase considerably.

CORTEX is an RTOS designed especially for embedded systems. It offers a flexible interrupt management system, preemptive and prioritized task scheduling with several scheduling policies, various memory management systems and synchronization primitives, and many other facilities.

Some issues that arose in the porting process due to incompatibility between the CORTEX design and the StarCore SC140 included:

- Stack alignment—the StarCore stack is aligned on 8-byte boundaries to allow parallel PUSH/POP instructions, while CORTEX assumes that stack is aligned on 4-byte boundaries. This problem was solved by adding extra code in the stack initialization routines to align the top and base of the stack as necessary.
- CORTEX assumes that all interrupts are listed in a single table, while the MSC8101 contains two interrupt tables (to accommodate two interrupt controllers). This problem was solved by concatenating the interrupt tables into one 128-entry table.

- StarCore's dual stack pointers (NSP and ESP) could not be used because CORTEX does not comply with StarCore's assumption that OS functions and interrupt handling routines return without switching tasks. CORTEX switches tasks inside the interrupt dispatcher and inside system calls, which requires that task stack is used.

Although CORTEX offers a stack tracing runtime-debugging feature, our port does not implement this feature because there is no support for stack tracing in the current release of the Enterprise C compiler.

Dynamically changed task priorities are not supported by CORTEX. However the application can change task priority at runtime.

Tasks can be dynamically created at runtime, but not statically. There is only one statically created task, the main task.

Available synchronization mechanisms include semaphores, mutexes, events, monitors and recursive resource locks, but not messages, mailboxes or queues.

Device drivers for the MSC8101 peripherals have not yet been written. This issue will be addressed in the next step of the project.

Because CORTEX is a complex system with many features, its memory footprint is quite large (56 KB). The software interrupt mechanism in CORTEX provides good flexibility for the interrupt system but it considerably slows task switching. To obtain higher performance figures, further optimization of the CORTEX source code, both platform independent and dependent, is necessary.

8 References

1. *MSC8101 Reference Manual*, order number MSC8101RM/D
2. *MSC8101 Programmer's Quick Reference*, order number MSC8101PG/D
3. *SC140 DSP Core Reference Manual*, order number MNSC140CORE/D
4. *SC100 Assembly Language Tools User's Manual*, order number MNSC100ALT/D
5. *SC100 Application Binary Interface Reference Manual*, order number MNSC100ABI/D
6. *SC100 C Compiler User's Manual*, order number MNSC100CC/D
7. *CodeWarrior Metrowerks Enterprise C Compiler User's Manual*
8. *Product Requirments for the StarCore SC140 RTOS* (Motorola internal document), Ed Martinez
9. CORTEX—documentation and sources, http://www.artesys.com.au
10. *Programming the MSC8101 Periodic Interrupt Timer (PIT)*, order number AN2144/D, Joe Liccese
11. *An Embedded Software Primer*, David E. Simon, Addison Wesley, 2000
12. *Modern Operating System*, Second edition, Andrew S. Tanenbaum, Prentice Hall, 2001
13. *Operating systems: Design and Implementation*, Second edition, Andrew S. Tanenbaum, Albert S. Woodhull, Prentice Hall, 1997